

# MONGO DB

## Lezione II

# RIPASSO

# RESOCONTO ESERCIZIO

# AND THE WINNING TEAM IS

Voto basato sull'esercizio!

1 - I Budini - 3/3 + - 3 pt

2 - I Mongolini, I Pavesini , Scobyboodidoo, Gengis khan - 3/3 - 2 pt

# AND THE WINNING CLASS IS

Voto basato sul test iniziale e sulla media degli esercizi!

1 - Pixel (68 esame / Esercizi 3/3) - 1 pt

2 - Zorin (64 esame / Esercizi 3/3) - 0 pt

# Bson

BSON è un formato informatico di interscambio dati creata per MongoDB. È la versione codificata in binario dei file JSON (JavaScript Object Notation), nata nel 2009 per il database MongoDB, anche se può essere utilizzata in modo indipendente, anche al di fuori di MongoDB.

Si tratta di un formato binario per strutture dati semplici e array associativi (chiamati oggetti o documenti in MongoDB). Può essere implementato nei principali linguaggi di programmazione, come Java, Python, Ruby, PHP, C, C++ e C#, JavaScript, Scala, Go e Swift.

# Bson

La differenza tra JSON e BSON è che BSON è occupa meno spazio di archiviazione ed è più veloce. Tuttavia, in alcuni casi, BSON può diventare meno efficiente del formato JSON a causa delle lunghezze fisse imposte e degli indici di array espliciti.

Esistono anche altre differenze tra JSON e BSON, in particolare per quanto concerne il tipo di dati supportati.

JSON supporta infatti solo i tipi di dati supportati da javaScript, BSON supporta anche altri dati (come ad esempio le date)

# Bson

Limiti da tenere bene a mente:

1. Un documento Bson di Mongo DB è limitato ad una dimensione massima di 16 megaByte.
2. Un documento Bson di Mongo DB è limitato ad un massimo di 100 livelli di nesting di documenti.

Questi 2 punti sono di fondamentale importanza per la struttura che andremo a creare nel nostro DB.

Creeremo una sola collection con elementi nestati? Creeremo più collection con i dati disseminati?

Seppur possano sembrare banali questi punti ci forzano ad un ragionamento particolarmente approfondito, specialmente in fase di creazione del DB!

# UpdateMany

Come abbiamo visto nella lezione precedente il comando preposto all'aggiornamento/modifica di un documento è UpdateOne. Questo comando permette di aggiornare un documento alla volta.

UpdateMany esegue la stessa operazione, con la differenza che invece di modificare un singolo documento modifica tutti quelli risultanti dalla query di filtro.

```
db.nomeCollection.updateMany(  
    {filtro},  
    {  
        $set: {documento}  
    }  
)
```

# Update VS Replace

Un comando molto simile all'**update** è il **replace**.

Questo funziona in modo similare, con la differenza che il documento non viene aggiornato in base ai parametri indicati ma direttamente sovrascritto.

L'`_id` non viene coinvolto in questa operazione.

```
db.nomeCollection.replaceOne(  
  {filtro},  
  {  
    nuovo_documento  
  }  
)
```

# Gli operatori di MongoDB

Gli operatori sono simboli speciali che aiutano i compilatori a eseguire operazioni matematiche o logiche. MongoDB offre diversi tipi di operatori per interagire con il database.

Gli operatori sono indicati dal simbolo \$

Esistono nove tipi di operatori, ognuno dei quali prende il nome dalla sua funzione. Per esempio, gli operatori logici utilizzano operazioni logiche. Per eseguirli, dovete utilizzare una parola chiave specifica e seguire la sintassi.

# Gli operatori di MongoDB

Vediamo qualche esempio! Gli operatori logici sono spesso utilizzati per filtrare i dati in base alle condizioni date.

## \$and

Una condizione “and” esegue un’operazione logica “and” su un array di due o più espressioni. Seleziona i documenti in cui tutte le condizioni delle espressioni sono soddisfatte.

Sintassi: { \$and: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }

Esempio: db.inventory.find( { \$and: [ { quantity: { \$lt: 15 } }, { price: 10 } ] } )

## \$or

Una condizione “or” esegue un’operazione logica “or” su un array di due o più espressioni. Seleziona i documenti in cui almeno una delle espressioni è vera.

Sintassi: { \$or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }

Esempio: db.inventory.find( { \$or: [ { quantity: { \$lt: 15 } }, { price: 10 } ] } )

# Gli operatori di MongoDB

E tutti gli altri operatori?

Bella domanda!

Siete arrivati al secondo anno e mi aspetto da voi più ricerca autonoma!

AKA → Google & Stack Overflow saranno i vostri migliori amici quest'anno!

# Find e cursori

Nella lezione precedente abbiamo visto il comando **find**, che (in maniera simile ad una SELECT di MySQL), restituisce un set di dati in base a criteri da noi espressi.

Una cosa particolare da notare è che questo comando non restituisce tutti i dati ma un cosiddetto oggetto “cursore”.

Vediamo l'esempio (trovate il file “Lista studenti.json nella cartella drive).

Inserite tutti gli studenti con una insertMany e successivamente lanciate il comando find()

Mongo DB(questa cosa non è limitata solo alla shell) non restituisce infatti un array con tutti i documenti (immaginate un DB con milioni di righe) ma bensì un cursore.

Questo ci permette di non elaborare tutti i dati in una volta sola ma di “spacchettare” il carico in svariate operazioni più piccole.

Per avere il risultato come unico array basta usare il comando **toArray**

# Find e cursori

Questo cursore può essere inserito in un ciclo forEach.

```
db.studenti.find().forEach((studente) =>  
{printjson(EJSON.stringify(studente))})
```

La shell utilizza Javascript, possiamo quindi passare una arrow function dentro questo ciclo e (per esempio) stampare ogni studente.

NOTA BENE → Ogni driver ha una sintassi diversa! Questi sono esempi utilizzando Mongosh, se ti trovi a dover usare driver specifici controlla bene la sintassi nella documentazione!

RICORDA → **find** non restituisce direttamente il documento ma un cursore di dati!  
**findOne** contrariamente restituisce direttamente un oggetto.

# Projection

Come abbiamo visto nel punto precedente il comando **find** restituisce tutto il contenuto del documento.

Non è sempre necessario estrarre tutto il documento però.

Tramite la clausola di Projection possiamo infatti selezionare solo i campi del documento che ci interessano.

```
db.nomeCollection.find(  
  {filtro},  
  {  
    campo_da_estrarre: 1,  
    campo_da_omettere: 0  
  })
```

# Gli schemi di Mongo DB

Mongo DB non supporta gli schemi, è infatti famoso per essere schema-less.

Quindi non abbiamo bisogno di uno schema, giusto?

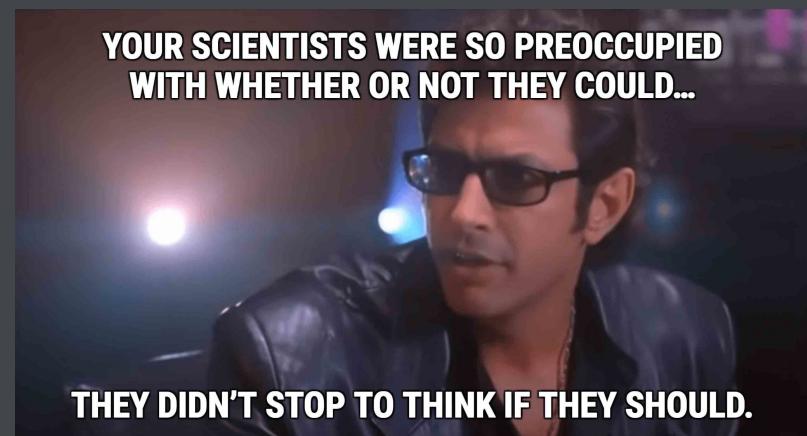
Si e no

# Schemas: la grande bugia

Abbiamo più volte detto che MongoDB e le sue collection non necessitano di uno schema definito per funzionare e questo è tecnicamente esatto.

Nella realtà della pratica però questa non è effettivamente la situazione più comune.

Il fatto che non siete obbligati/forzati ad avere uno schema non implica che usarli sia sbagliato!



# Schemas e come approcciarli

Possiamo definire 3 principali approcci.

Vediamo degli esempi per una collection di prodotti di uno shop online:

Caos  
Nessuno schema

```
{  
  "titolo" : "Lo Hobbit",  
  "prezzo": 13.50  
}  
  
{  
  "nome" : "Segnalibro",  
  "disponibilita" : true  
}
```

Via di mezzo  
Schema parziale

```
{  
  "nome" : "Lo Hobbit",  
  "prezzo": 13.50,  
  "disponibilita" : true  
}  
  
{  
  "nome" : "Segnalibro",  
  "disponibilita" : true  
}
```

SQL  
Schema completo

```
{  
  "nome" : "Lo Hobbit",  
  "prezzo": 13.50,  
  "disponibilita" : true  
}  
  
{  
  "nome" : "Segnalibro",  
  "prezzo": null,  
  "disponibilita" : true  
}
```

# Schemas e Data Modelling

Come decidere il modo migliore di strutturare i miei schemi (teorici o forzati che siano) e più ampiamente come creare un DB che soddisfi le nostre esigenze?

Che dati salverò o dovrò generare?

Informazioni utente, informazioni prodotti, ordini

Definisce i campi di cui avremo bisogno e le loro relazioni

Dove userò questi dati?

Landing page, shop online, cataloghi

Definisce le collezione e il raggruppamento dei dati

Che informazioni/dati mostrerò?

Landing page, shop online, cataloghi

Definisce le query che ci serviranno

Quanto spesso dovrò recuperare questi dati dal DB?

Ad ogni ricarica pagina, ogni tot secondi

Definisce se dobbiamo considerare l'ottimizzazione dei dati per un aggiornamento veloce

Quanto spesso dovrò scrivere questi dati sul DB?

Ordini → Spesso  
Dati del prodotto → Raramente

Definisce se dobbiamo considerare l'ottimizzazione dei dati per una scrittura veloce

# RICORDA!

Mongo DB è orientato ai documenti e consequenzialmente agli oggetti!

L'obiettivo dovrebbe sempre essere salvare i dati nel formato che poi ti servirà nella tua applicazione/sito/programma!

# Le relazioni (di nuovo)

1 a 1

1 a molti

molti a molti

# Le relazioni - considerazioni

Nestate / Embeddate

{

Studenti

```
nome: "Marco",
cognome: "Tomelleri"
indirizzo: {
    via: "Pinco panco",
    cap: 37666
}
```

}

Riferimenti

Clienti

```
nome: "Marco",
libri_preferiti:[..., ...]
```

{

Clienti

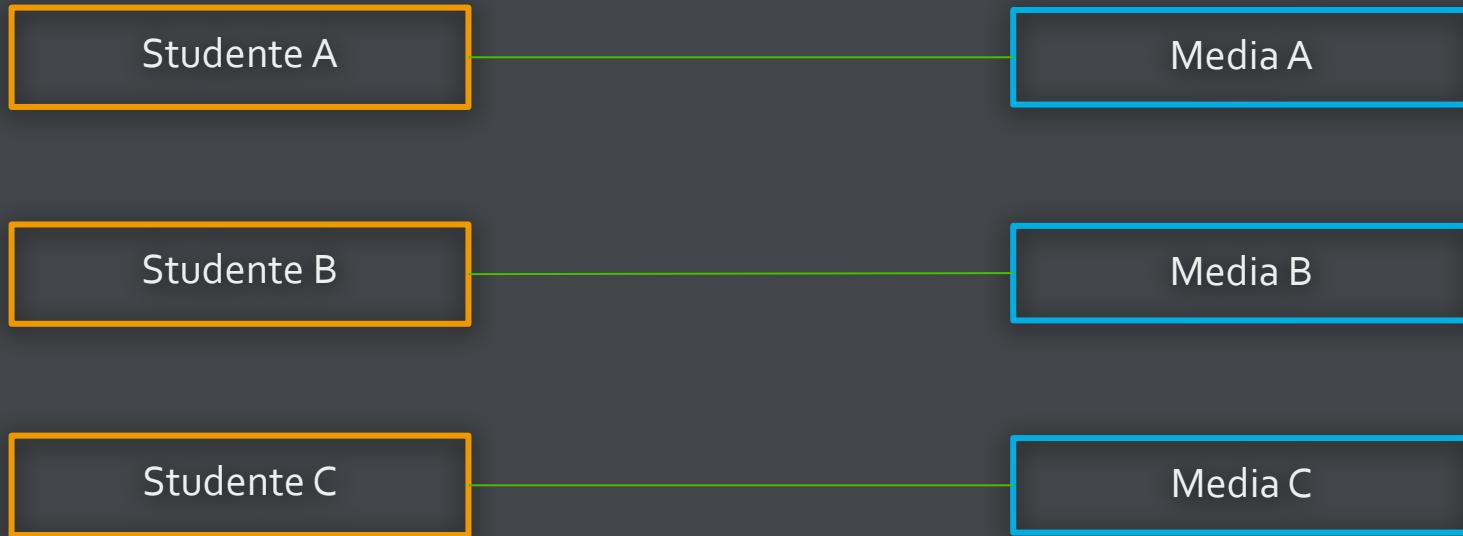
```
nome: "Marco",
libri_preferiti:[‘id1’, ‘id2’]
```

{

# Relazione uno a uno nestata

Esempio → Studente e Media voti

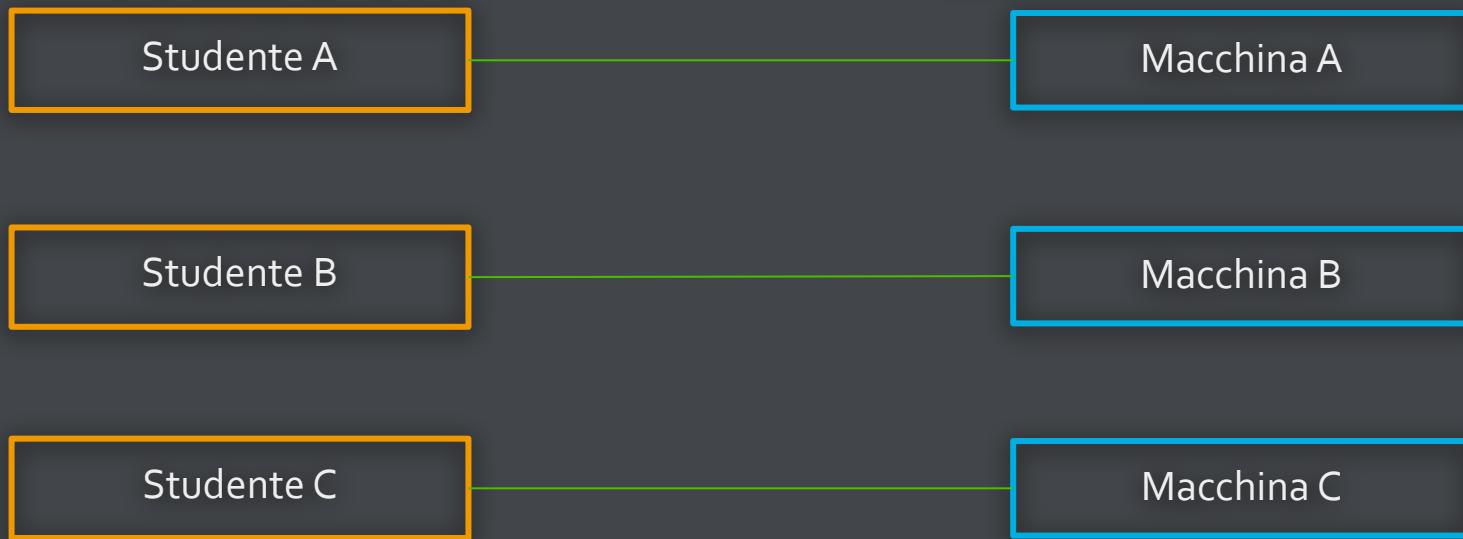
Uno studente ha una media di voti e la media dei suoi voti appartiene solo a quello studente



# Relazione uno a uno relazionata

Esempio → Studente e Macchina

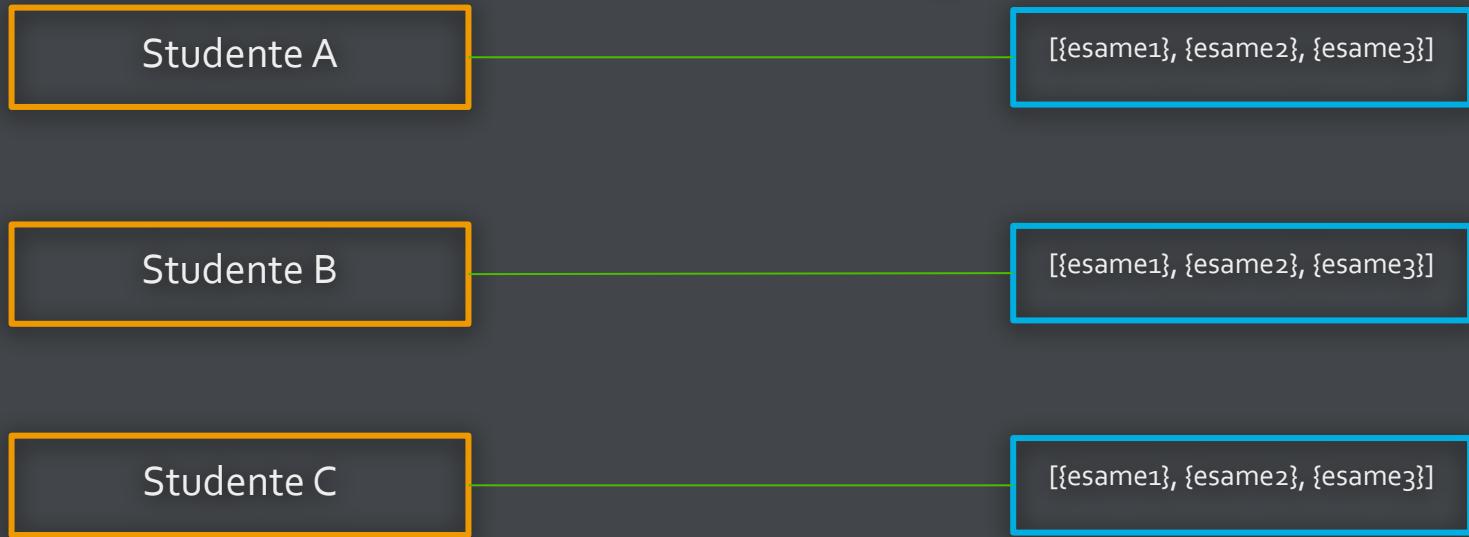
Uno studente ha una macchina e una macchina appartiene ad uno studente



# Relazione uno a molti nestata

Esempio → Studente e risultati esami materia

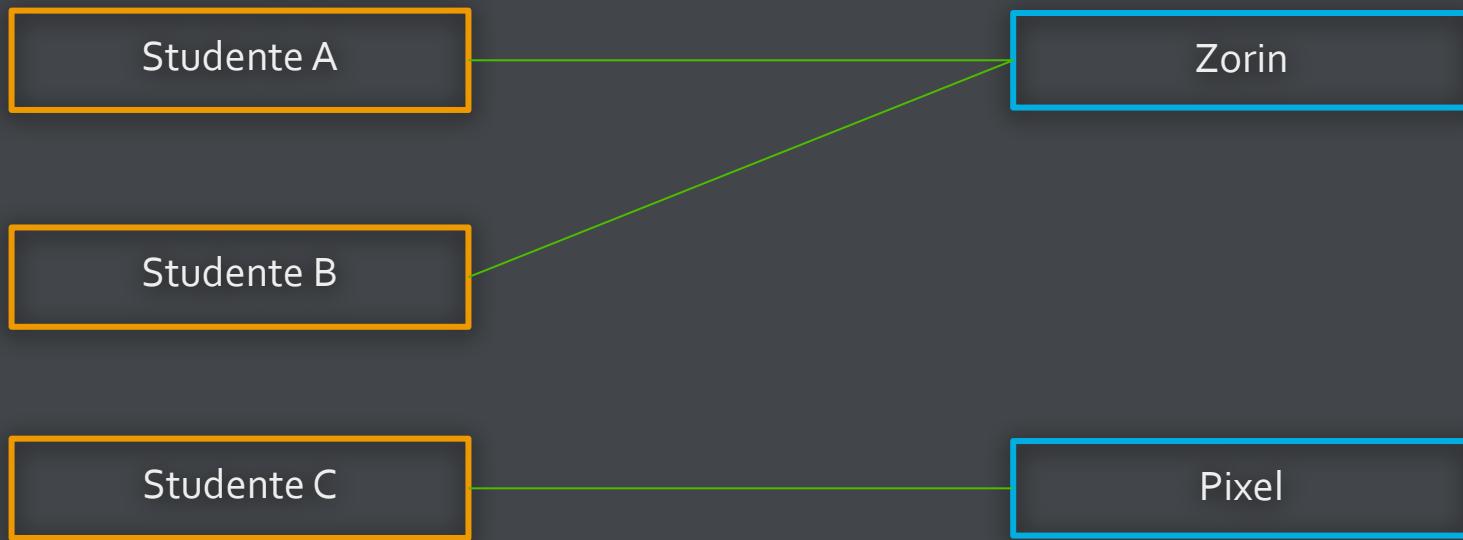
Uno studente ha un set di voti (array) e quell'array appartiene solo a quello studente



# Relazione uno a molti relazionata

Esempio → Studente e classe

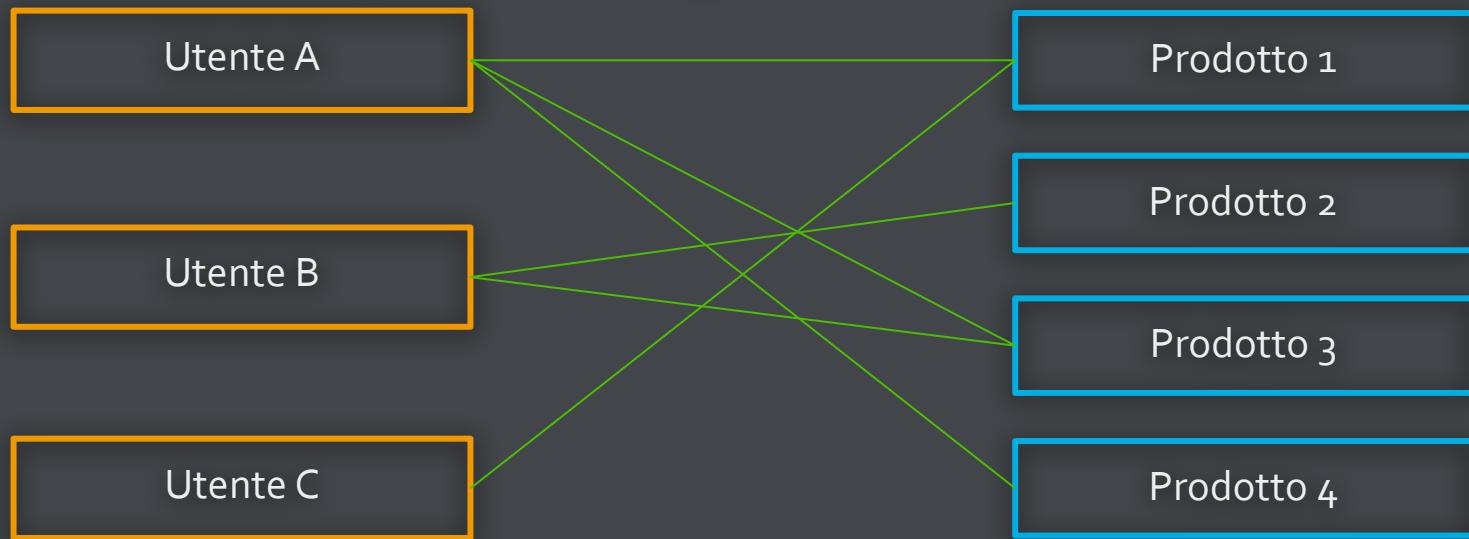
Uno studente ha una classe e una classe ha molti studenti



# Relazione molti a molti nestata

Esempio → Utenti e ordini prodotti

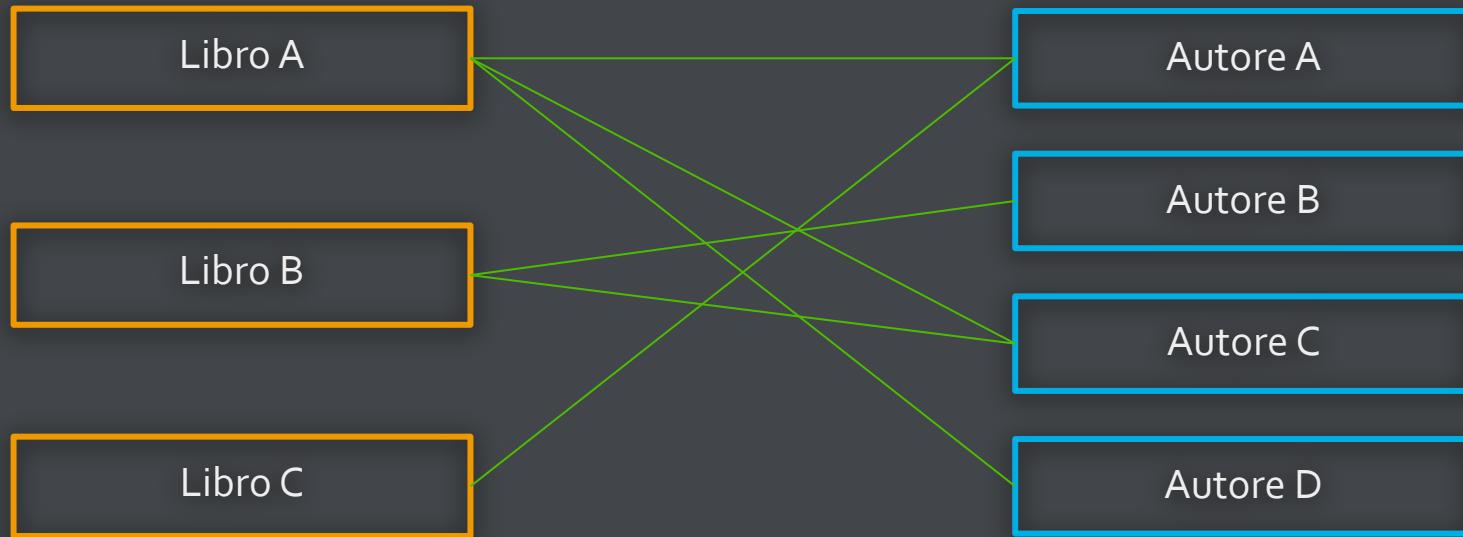
Un utente può ordinare molti prodotti sono ordinati da molti utenti



# Relazione molti a molti relazionata

Esempio → Libri e autori

Un libro ha molti autori e un autore ha scritto molti libri



# Le relazioni - riassunto

## Nestate / Embeddate

Dati raggruppati logicamente

Ottimo per dati che appartengono alla stessa entità e che riducono la duplicazione di dati

Attenzione al limite di 100 nesting o ad array troppo grandi (limite 16 megabyte)

## Riferimenti

Dati sparsi in più collection

Ottimo per gestire le relazioni ed evitare i duplicati

Permette di bypassare il limite di nesting e di dimensione suddividendo in più documenti

Non c'è una regola precisa ma se tenete in considerazione i punti precedenti e vi chiedete:  
Quanto spesso uso i miei dati?, Come li uso?, Quanto spesso li cambio? Come sono relazionati?  
Riuscirete a creare un DB che soddisfi le esigenze

# Introduzione ai Join

Anche in MongoDB possiamo effettuare JOIN, seppur in maniera diversa da SQL.

Per effettuare questa operazione utilizzeremo il comando **aggregate**.

Questo comando ha molteplici funzioni ma per ora ci limiteremo al suo utilizzo di join con l'operatore **\$lookUp**.

Vediamo la sintassi →

```
db.collectionDiPartenza.aggregate([
  {$lookup:{from:"collectionDiDestinazione",
    localField:"id_CDP",
    foreignField:"id_CDD",
    as:"aliasCDD" }
  }
]);
```

# MONGO DB

## Lezione II

### Esercizio I

# Esercizio I

Vedi link